carrierwaveuploader / carrierwave

Watch 157 Star 7,017 Fork 1,248

Classier solution for file uploads for Rails, Sinatra and other Ruby web frameworks https://github.com/carrierwaveuploader/carrierwave

2,068 commits 7 branches 46 releases 283 contributors

Branch: master New pull request Find file Clone or download

mshibuya committed on GitHub Merge pull request #2041 from cfcosta/remove-warnings-from-specs ... Latest commit ec85083 5 days ago

features	Replace ShamRack by WebMock in specs	9 months ago
gemfiles	Add Rails 5.0 stable gemfile for testing.	21 days ago
lib	Merge pull request #2036 from t-oginogin/fix_difference_cache_dir_nam...	7 days ago
script	Converted to use Hoe	7 years ago
spec	Remove deprecated opt-in flag from ActiveRecord 5	6 days ago
.gitignore	Added *_copy fixtures to gitignore	7 months ago
.rubocop.yml	Add a `rubocop.yml` file with basic cops enabled	9 months ago
.travis.yml	Add Rails 5.0 stable gemfile for testing.	21 days ago
CHANGELOG.md	Version 1.0.0.beta	a month ago
CONTRIBUTING.md	Update Ruby version from CONTRIBUTING.md	10 months ago
Gemfile	Add activemodel-serializers-xml to Rails 5 builds	2 months ago
README.md	Update README.md	25 days ago
Rakefile	Remove UTF-8 magic comments from all the Ruby files	10 months ago
carrierwave.gemspec	Upgrade testing tools	8 months ago
cucumber.yml	no more more/core division, just one single gem	8 years ago

README.md

CarrierWave

This gem provides a simple and extremely flexible way to upload files from Ruby applications. It works well with Rack based web applications, such as Ruby on Rails.

build passing | code climate 3.5 | MIT | 13/13

> ## carrierwave version disclaimer
>
> ***This README is for a branch which is still in development. Please switch to latest `0.x` branch for stable version.***

Information

- RDoc documentation available on RubyDoc.info

- Source code available on GitHub
- More information, known limitations, and how-tos available on the wiki

Getting Help

- Please ask the community on Stack Overflow for help if you have any questions. Please do not post usage questions on the issue tracker.
- Please report bugs on the issue tracker but read the "getting help" section in the wiki first.

Installation

Install the latest release:

```
$ gem install carrierwave -v "1.0.0.beta"
```

In Rails, add it to your Gemfile:

```
gem 'carrierwave', '>= 1.0.0.beta', '< 2.0'
```

Finally, restart the server to apply the changes.

As of version 1.0.0 (forthcoming), CarrierWave requires Rails 4.0 or higher and Ruby 2.0 or higher. If you're on Rails 3, you should use v0.11.0.

Getting Started

Start off by generating an uploader:

```
rails generate uploader Avatar
```

this should give you a file in:

```
app/uploaders/avatar_uploader.rb
```

Check out this file for some hints on how you can customize your uploader. It should look something like this:

```
class AvatarUploader < CarrierWave::Uploader::Base
  storage :file
end
```

You can use your uploader class to store and retrieve files like this:

```
uploader = AvatarUploader.new

uploader.store!(my_file)

uploader.retrieve_from_store!('my_file.png')
```

CarrierWave gives you a `store` for permanent storage, and a `cache` for temporary storage. You can use different stores, including filesystem and cloud storage.

Most of the time you are going to want to use CarrierWave together with an ORM. It is quite simple to mount uploaders on columns in your model, so you can simply assign files and get going:

ActiveRecord

Make sure you are loading CarrierWave after loading your ORM, otherwise you'll need to require the relevant extension manually, e.g.:

```
require 'carrierwave/orm/activerecord'
```

Add a string column to the model you want to mount the uploader by creating a migration:

```
rails g migration add_avatar_to_users avatar:string
rake db:migrate
```

Open your model file and mount the uploader:

```
class User < ActiveRecord::Base
  mount_uploader :avatar, AvatarUploader
end
```

Now you can cache files by assigning them to the attribute, they will automatically be stored when the record is saved.

```
u = User.new
u.avatar = params[:file] # Assign a file like this, or

# like this
File.open('somewhere') do |f|
  u.avatar = f
end

u.save!
u.avatar.url # => '/url/to/file.png'
u.avatar.current_path # => 'path/to/file.png'
u.avatar_identifier # => 'file.png'
```

Note: `u.avatar` will never return nil, even if there is no photo associated to it. To check if a photo was saved to the model, use `u.avatar.file.nil?` instead.

DataMapper, Mongoid, Sequel

Other ORM support has been extracted into separate gems:

- carrierwave-datamapper
- carrierwave-mongoid
- carrierwave-sequel

There are more extensions listed in the wiki

Multiple file uploads

Note: You must specify using the master branch to enable this feature:

```
gem 'carrierwave', github: 'carrierwaveuploader/carrierwave'.
```

CarrierWave also has convenient support for multiple file upload fields.

ActiveRecord

Add a column which can store an array. This could be an array column or a JSON column for example. Your choice depends on what your database supports. For example, create a migration like this:

```
rails g migration add_avatars_to_users avatars:json
rake db:migrate
```

Open your model file and mount the uploader:

```
class User < ActiveRecord::Base
  mount_uploaders :avatars, AvatarUploader
end
```

Make sure your file input fields are set up as multiple file fields. For example in Rails you'll want to do something like this:

```
<%= form.file_field :avatars, multiple: true %>
```

Also, make sure your upload controller permits the multiple file upload attribute *pointing to an empty array in a hash*. For example:

```
params.require(:user).permit(:email, :first_name, :last_name, {avatars: []})
```

Now you can select multiple files in the upload dialog (e.g. SHIFT+SELECT), and they will automatically be stored when the record is saved.

```
u = User.new(params[:user])
u.save!
u.avatars[0].url # => '/url/to/file.png'
u.avatars[0].current_path # => 'path/to/file.png'
u.avatars[0].identifier # => 'file.png'
```

Changing the storage directory

In order to change where uploaded files are put, just override the `store_dir` method:

```
class MyUploader < CarrierWave::Uploader::Base
  def store_dir
    'public/my/upload/directory'
  end
end
```

This works for the file storage as well as Amazon S3 and Rackspace Cloud Files. Define `store_dir` as `nil` if you'd like to store files at the root level.

If you store files outside the project root folder, you may want to define `cache_dir` in the same way:

```
class MyUploader < CarrierWave::Uploader::Base
  def cache_dir
    '/tmp/projectname-cache'
  end
end
```

Securing uploads

Certain files might be dangerous if uploaded to the wrong location, such as PHP files or other script files. CarrierWave allows you to specify a whitelist of allowed extensions or content types.

If you're mounting the uploader, uploading a file with the wrong extension will make the record invalid instead. Otherwise, an error is raised.

```
class MyUploader < CarrierWave::Uploader::Base
  def extension_whitelist
    %w(jpg jpeg gif png )
  end
end
```

The same thing could be done using content types. Let's say we need an uploader that accepts only images. This can be done like this

```
class MyUploader < CarrierWave::Uploader::Base
  def content_type_whitelist
    /image\//
  end
end
```

You can use a blacklist to reject content types. Let's say we need an uploader that reject JSON files. This can be done like this

```
class NoJsonUploader < CarrierWave::Uploader::Base
  def content_type_blacklist
    ['application/text ', 'application/json ']
  end
end
```

Filenames and unicode chars

Another security issue you should care for is the file names (see Ruby On Rails Security Guide). By default, CarrierWave provides only English letters, arabic numerals and some symbols as white-listed characters in the file name. If you want to support local scripts (Cyrillic letters, letters with diacritics and so on), you have to override `sanitize_regexp` method. It should return regular expression which would match all *non*-allowed symbols.

```
CarrierWave::SanitizedFile.sanitize_regexp = /[^[:word:]\.\-\+]/
```

Also make sure that allowing non-latin characters won't cause a compatibility issue with a third-party plugins or client-side software.

Setting the content type

As of v0.11.0, the `mime-types` gem is a runtime dependency and the content type is set automatically. You no longer need to do this manually.

Adding versions

Often you'll want to add different versions of the same file. The classic example is image thumbnails. There is built in support for this*:

Note: You must have Imagemagick and MiniMagick installed to do image resizing. MiniMagick is a Ruby interface for Imagemagick which is a C program. This is why MiniMagick fails on 'bundle install' without Imagemagick installed.

Some documentation refers to RMagick instead of MiniMagick but MiniMagick is recommended.

To install Imagemagick on OSX with homebrew type the following:

```
$ brew install imagemagick
```

```
class MyUploader < CarrierWave::Uploader::Base
  include CarrierWave::MiniMagick
```

```
  process resize_to_fit: [800, 800]

  version :thumb do
    process resize_to_fill: [200,200]
  end

end
```

When this uploader is used, an uploaded image would be scaled to be no larger than 800 by 800 pixels. A version called thumb is then created, which is scaled and cropped to exactly 200 by 200 pixels. The uploader could be used like this:

```
uploader = AvatarUploader.new
uploader.store!(my_file)                                # size: 1024x768

uploader.url # => '/url/to/my_file.png'               # size: 800x600
uploader.thumb.url # => '/url/to/thumb_my_file.png'   # size: 200x200
```

One important thing to remember is that process is called *before* versions are created. This can cut down on processing cost.

It is possible to nest versions within versions:

```
class MyUploader < CarrierWave::Uploader::Base

  version :animal do
    version :human
    version :monkey
    version :llama
  end
end
```

Conditional versions

Occasionally you want to restrict the creation of versions on certain properties within the model or based on the picture itself.

```
class MyUploader < CarrierWave::Uploader::Base

  version :human, if: :is_human?
  version :monkey, if: :is_monkey?
  version :banner, if: :is_landscape?

private

  def is_human? picture
    model.can_program?( :ruby )
  end

  def is_monkey? picture
    model.favorite_food == 'banana'
  end

  def is_landscape? picture
    image = MiniMagick::Image.open(picture.path)
    image[ :width ] > image[ :height ]
  end

end
```

The `model` variable points to the instance object the uploader is attached to.

Create versions from existing versions

For performance reasons, it is often useful to create versions from existing ones instead of using the original file. If your uploader generates several versions where the next is smaller than the last, it will take less time to generate from a smaller, already processed image.

```
class MyUploader < CarrierWave::Uploader::Base

  version :thumb do
    process resize_to_fill: [280, 280]
  end

  version :small_thumb , from_version: :thumb do
    process resize_to_fill: [20, 20]
  end

end
```

The option `:from_version` uses the file cached in the `:thumb` version instead of the original version, potentially resulting in faster processing.

Making uploads work across form redisplays

Often you'll notice that uploaded files disappear when a validation fails. CarrierWave has a feature that makes it easy to remember the uploaded file even in that case. Suppose you `user` model has an uploader mounted on `avatar` file, just add a hidden field called `avatar_cache` (don't forget to add it to the attr_accessible list as necessary). In Rails, this would look like this:

```
<%= form_for @user , html: { multipart: true } do |f| %>
  <p>
    <label>My Avatar</ label>
    <%= f.file_field :avatar %>
    <%= f.hidden_field :avatar_cache %>
  </p>
<% end %>
```

It might be a good idea to show the user that a file has been uploaded, in the case of images, a small thumbnail would be a good indicator:

```
<%= form_for @user , html: { multipart: true } do |f| %>
  <p>
    <label>My Avatar</ label>
    <%= image_tag( @user .avatar_url) if @user .avatar? %>
    <%= f.file_field :avatar %>
    <%= f.hidden_field :avatar_cache %>
  </p>
<% end %>
```

Removing uploaded files

If you want to remove a previously uploaded file on a mounted uploader, you can easily add a checkbox to the form which will remove the file when checked.

```
<%= form_for @user , html: { multipart: true } do |f| %>
  <p>
    <label>My Avatar</ label>
    <%= image_tag( @user .avatar_url) if @user .avatar? %>
    <%= f.file_field :avatar %>
  </p>

  <p>
    <label>
      <%= f.check_box :remove_avatar %>
```

```
      Remove avatar
    </label>
  </p>
<% end %>
```

If you want to remove the file manually, you can call `remove_avatar!`, then save the object.

```
@user.remove_avatar!
@user.save
#=> true
```

Uploading files from a remote location

Your users may find it convenient to upload a file from a location on the Internet via a URL. CarrierWave makes this simple, just add the appropriate attribute to your form and you're good to go:

```
<%= form_for @user, html: { multipart: true } do |f| %>
  <p>
    <label>My Avatar URL:</label>
    <%= image_tag(@user.avatar_url) if @user.avatar? %>
    <%= f.text_field :remote_avatar_url %>
  </p>
<% end %>
```

If you're using ActiveRecord, CarrierWave will indicate invalid URLs and download failures automatically with attribute validation errors. If you aren't, or you disable CarrierWave's `validate_download` option, you'll need to handle those errors yourself.

Providing a default URL

In many cases, especially when working with images, it might be a good idea to provide a default url, a fallback in case no file has been uploaded. You can do this easily by overriding the `default_url` method in your uploader:

```
class MyUploader < CarrierWave::Uploader::Base
  def default_url(*args)
    "/images/fallback/" + [version_name, "default.png"].compact.join('_')
  end
end
```

Or if you are using the Rails asset pipeline:

```
class MyUploader < CarrierWave::Uploader::Base
  def default_url(*args)
    ActionController::Base.helpers.asset_path("fallback/" + [version_name, "default.png"].compact.join('_'))
  end
end
```

Recreating versions

You might come to a situation where you want to retroactively change a version or add a new one. You can use the `recreate_versions!` method to recreate the versions from the base file. This uses a naive approach which will re-upload and process the specified version or all versions, if none is passed as an argument.

When you are generating random unique filenames you have to call `save!` on the model after using `recreate_versions!`. This is necessary because `recreate_versions!` doesn't save the new filename to the database. Calling `save!` yourself will prevent that the database and file system are running out of sync.

```
instance = MyUploader.new
instance.recreate_versions!(:thumb, :large)
```

Or on a mounted uploader:

```
User.find_each do |user|
  user.avatar.recreate_versions!
end
```

Note: `recreate_versions!` will throw an exception on records without an image. To avoid this, scope the records to those with images or check if an image exists within the block. If you're using ActiveRecord, recreating versions for a user avatar might look like this:

```
User.find_each do |user|
  user.avatar.recreate_versions! if user.avatar?
end
```

Configuring CarrierWave

CarrierWave has a broad range of configuration options, which you can configure, both globally and on a per-uploader basis:

```
CarrierWave.configure do |config|
  config.permissions = 0666
  config.directory_permissions = 0777
  config.storage = :file
end
```

Or alternatively:

```
class AvatarUploader < CarrierWave::Uploader::Base
  permissions 0777
end
```

If you're using Rails, create an initializer for this:

```
config/initializers/carrierwave.rb
```

If you want CarrierWave to fail noisily in development, you can change these configs in your environment file:

```
CarrierWave.configure do |config|
  config.ignore_integrity_errors = false
  config.ignore_processing_errors = false
  config.ignore_download_errors = false
end
```

Testing with CarrierWave

It's a good idea to test your uploaders in isolation. In order to speed up your tests, it's recommended to switch off processing in your tests, and to use the file storage. In Rails you could do that by adding an initializer with:

```
if Rails.env.test? or Rails.env.cucumber?
  CarrierWave.configure do |config|
    config.storage = :file
    config.enable_processing = false
```

```
  end
end
```

Remember, if you have already set `storage :something` in your uploader, the `storage` setting from this initializer will be ignored.

If you need to test your processing, you should test it in isolation, and enable processing only for those tests that need it.

CarrierWave comes with some RSpec matchers which you may find useful:

```
require 'carrierwave/test/matchers'

describe MyUploader do
  include CarrierWave::Test::Matchers

  let(:user) { double('user') }
  let(:uploader) { MyUploader.new(user, :avatar) }

  before do
    MyUploader.enable_processing = true
    File.open(path_to_file) { |f| uploader.store!(f) }
  end

  after do
    MyUploader.enable_processing = false
    uploader.remove!
  end

  context 'the thumb version' do
    it "scales down a landscape image to be exactly 64 by 64 pixels" do
      expect(uploader.thumb).to have_dimensions(64, 64)
    end
  end

  context 'the small version' do
    it "scales down a landscape image to fit within 200 by 200 pixels" do
      expect(uploader.small).to be_no_larger_than(200, 200)
    end
  end

  it "makes the image readable only to the owner and not executable" do
    expect(uploader).to have_permissions(0600)
  end

  it "has the correct format" do
    expect(uploader).to be_format('png')
  end
end
```

Setting the enable_processing flag on an uploader will prevent any of the versions from processing as well. Processing can be enabled for a single version by setting the processing flag on the version like so:

```
@uploader.thumb.enable_processing = true
```

Fog

If you want to use fog you must add in your CarrierWave initializer the following lines

```
config.fog_provider = 'fog' # 'fog/aws' etc. Defaults to 'fog'
config.fog_credentials = { ... } # Provider specific credentials
```

Using Amazon S3

Fog AWS is used to support Amazon S3. Ensure you have it in your Gemfile:

```
gem "fog-aws"
```

You'll need to provide your fog_credentials and a fog_directory (also known as a bucket) in an initializer. For the sake of performance it is assumed that the directory already exists, so please create it if it needs to be. You can also pass in additional options, as documented fully in lib/carrierwave/storage/fog.rb. Here's a full example:

```
CarrierWave.configure do |config|
  config.fog_provider = 'fog/aws'                         # required
  config.fog_credentials = {
    provider:               'AWS',                        # required
    aws_access_key_id:      'xxx',                        # required
    aws_secret_access_key:  'yyy',                        # required
    region:                 'eu-west-1',                  # optional, defaults to 'us-east-1'
    host:                   's3.example.com',             # optional, defaults to nil
    endpoint:               'https://s3.example.com:8080' # optional, defaults to nil
  }
  config.fog_directory    = 'name_of_directory'                          # required
  config.fog_public       = false                                        # optional, defaults to true
  config.fog_attributes   = { 'Cache-Control' => "max-age=#{365.day.to_i}" } # optional, defaults to {}
end
```

In your uploader, set the storage to :fog

```
class AvatarUploader < CarrierWave::Uploader::Base
  storage :fog
end
```

That's it! You can still use the `CarrierWave::Uploader#url` method to return the url to the file on Amazon S3.

Using Rackspace Cloud Files

Fog is used to support Rackspace Cloud Files. Ensure you have it in your Gemfile:

```
gem "fog"
```

You'll need to configure a directory (also known as a container), username and API key in the initializer. For the sake of performance it is assumed that the directory already exists, so please create it if need be.

Using a US-based account:

```
CarrierWave.configure do |config|
  config.fog_provider = "fog/rackspace/storage"   # optional, defaults to "fog"
  config.fog_credentials = {
    provider:             'Rackspace',
    rackspace_username:   'xxxxxx',
    rackspace_api_key:    'yyyyyy',
    rackspace_region:     :ord                    # optional, defaults to :dfw
  }
  config.fog_directory = 'name_of_directory'
end
```

Using a UK-based account:

```
CarrierWave.configure do |config|
  config.fog_provider = "fog/rackspace/storage"   # optional, defaults to "fog"
  config.fog_credentials = {
    provider:             'Rackspace',
```

```
    rackspace_username:   'xxxxxx',
    rackspace_api_key:    'yyyyyy',
    rackspace_auth_url:   Fog::Rackspace::UK_AUTH_ENDPOINT,
    rackspace_region:     :lon
  }
  config.fog_directory  = 'name_of_directory'
end
```

You can optionally include your CDN host name in the configuration. This is *highly* recommended, as without it every request requires a lookup of this information.

```
config.asset_host  = "http://c000000.cdn.rackspacecloud.com"
```

In your uploader, set the storage to :fog

```
class AvatarUploader < CarrierWave::Uploader::Base
  storage :fog
end
```

That's it! You can still use the `CarrierWave::Uploader#url` method to return the url to the file on Rackspace Cloud Files.

Using Google Storage for Developers

Fog is used to support Google Storage for Developers. Ensure you have it in your Gemfile:

```
gem "fog-google"
gem "google-api-client", ">= 0.6.2", "< 0.9"
gem "mime-types"
```

You'll need to configure a directory (also known as a bucket), access key id and secret access key in the initializer. For the sake of performance it is assumed that the directory already exists, so please create it if need be.

Please read the fog-google README on how to get credentials.

```
CarrierWave.configure do |config|
  config.fog_provider   = 'fog/google'                          # required
  config.fog_credentials   = {
    provider:                         'Google',
    google_storage_access_key_id:     'xxxxxx',
    google_storage_secret_access_key: 'yyyyyy'
  }
  config.fog_directory  = 'name_of_directory'
end
```

In your uploader, set the storage to :fog

```
class AvatarUploader < CarrierWave::Uploader::Base
  storage :fog
end
```

That's it! You can still use the `CarrierWave::Uploader#url` method to return the url to the file on Google.

Optimized Loading of Fog

Since Carrierwave doesn't know which parts of Fog you intend to use, it will just load the entire library (unless you use e.g. `[fog-aws, fog-google]` instead of fog proper). If you prefer to load fewer classes into your application, you need to load those parts of Fog yourself *before* loading CarrierWave in your Gemfile. Ex:

```
gem "fog", "~> 1.27", require: "fog/rackspace/storage"
gem "carrierwave"
```

A couple of notes about versions:

- This functionality was introduced in Fog v1.20.
- This functionality is slated for CarrierWave v1.0.0.

If you're not relying on Gemfile entries alone and are requiring "carrierwave" anywhere, ensure you require "fog/rackspace/storage" before it. Ex:

```
require "fog/rackspace/storage"
require "carrierwave"
```

Beware that this specific require is only needed when working with a fog provider that was not extracted to its own gem yet. A list of the extracted providers can be found in the page of the `fog` organizations here.

When in doubt, inspect `Fog.constants` to see what has been loaded.

Dynamic Asset Host

The `asset_host` config property can be assigned a proc (or anything that responds to `call`) for generating the host dynamically. The proc-compliant object gets an instance of the current `CarrierWave::Storage::Fog::File` or `CarrierWave::SanitizedFile` as its only argument.

```
CarrierWave.configure do |config|
  config.asset_host = proc do |file|
    identifier = # some logic
    "http://#{identifier}.cdn.rackspacecloud.com"
  end
end
```

Using RMagick

If you're uploading images, you'll probably want to manipulate them in some way, you might want to create thumbnail images for example. CarrierWave comes with a small library to make manipulating images with RMagick easier, you'll need to include it in your Uploader:

```
class AvatarUploader < CarrierWave::Uploader::Base
  include CarrierWave::RMagick
end
```

The RMagick module gives you a few methods, like `CarrierWave::RMagick#resize_to_fill` which manipulate the image file in some way. You can set a `process` callback, which will call that method any time a file is uploaded. There is a demonstration of convert here. Convert will only work if the file has the same file extension, thus the use of the filename method.

```
class AvatarUploader < CarrierWave::Uploader::Base
  include CarrierWave::RMagick

  process resize_to_fill: [200, 200]
  process convert: 'png'

  def filename
    super.chomp(File.extname(super)) + '.png' if original_filename.present?
  end
end
```

Check out the manipulate! method, which makes it easy for you to write your own manipulation methods.

Using MiniMagick

MiniMagick is similar to RMagick but performs all the operations using the 'mogrify' command which is part of the standard ImageMagick kit. This allows you to have the power of ImageMagick without having to worry about installing all the RMagick libraries.

See the MiniMagick site for more details:

https://github.com/minimagick/minimagick

And the ImageMagick command line options for more for whats on offer:

http://www.imagemagick.org/script/command-line-options.php

Currently, the MiniMagick carrierwave processor provides exactly the same methods as for the RMagick processor.

```
class AvatarUploader < CarrierWave::Uploader::Base
  include CarrierWave::MiniMagick

  process resize_to_fill: [200, 200]
end
```

Migrating from Paperclip

If you are using Paperclip, you can use the provided compatibility module:

```
class AvatarUploader < CarrierWave::Uploader::Base
  include CarrierWave::Compatibility::Paperclip
end
```

See the documentation for `CarrierWave::Compatibility::Paperclip` for more details.

Be sure to use mount_on to specify the correct column:

```
mount_uploader :avatar, AvatarUploader, mount_on: :avatar_file_name
```

I18n

The Active Record validations use the Rails i18n framework. Add these keys to your translations file:

```
errors:
  messages:
    carrierwave_processing_error:  "Cannot resize image. "
    carrierwave_integrity_error:  "Not an image. "
    carrierwave_download_error:  "Couldn't download image. "
    extension_whitelist_error:  "You are not allowed to upload %{extension} files, allowed types: %{allowed_types}
    extension_blacklist_error:  "You are not allowed to upload %{extension} files, prohibited types: %{prohibited_ty
```

Large files

By default, CarrierWave copies an uploaded file twice, first copying the file into the cache, then copying the file into the store. For large files, this can be prohibitively time consuming.

You may change this behavior by overriding either or both of the `move_to_cache` and `move_to_store` methods:

```
class MyUploader < CarrierWave::Uploader::Base
  def move_to_cache
    true
  end

  def move_to_store
    true
  end
end
```

When the `move_to_cache` and/or `move_to_store` methods return true, files will be moved (instead of copied) to the cache and store respectively.

This has only been tested with the local filesystem store.

Skipping ActiveRecord callbacks

By default, mounting an uploader into an ActiveRecord model will add a few callbacks. For example, this code:

```
class User
  mount_uploader :avatar, AvatarUploader
end
```

Will add these callbacks:

```
after_save :store_avatar!
before_save :write_avatar_identifier
after_commit :remove_avatar!, on: :destroy
after_commit :mark_remove_avatar_false, on: :update
after_save :store_previous_changes_for_avatar
after_commit :remove_previously_stored_avatar, on: :update
```

If you want to skip any of these callbacks (eg. you want to keep the existing avatar, even after uploading a new one), you can use ActiveRecord's `skip_callback` method.

```
class User
  mount_uploader :avatar, AvatarUploader
  skip_callback :commit, :after, :remove_previously_stored_avatar
end
```

Contributing to CarrierWave

See CONTRIBUTING.md

License

Copyright (c) 2008-2015 Jonas Nicklas

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,

WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

© 2016 GitHub, Inc. Terms Privacy Security Status Help

Contact GitHub API Training Shop Blog About